lab6实验报告

一、代码实现

对七个机器码/汇编语言翻译为C语言，代码如下

1.lab0l

```
short lab0l(short r0, short r1)
{
    short r7 = 0;
    for(; r1 != 0; r1--)
    {
        r7 += r0;
    }
    return r7;
}
```

- 空间复杂度：S(1);
- 时间复杂度：O(n); 取决于乘数的大小。对顺序很敏感，例如5*4000，执行4000次，若交换位置只要5次。

2.lab0p

```
short lab0p(short r0, short r1)
{
    short r2, r3, r7 = 0;
    for(r2 = 1; r2 != 0; r2 *= 2)
    {
        r3 = r2 & r1;
        if(r3)
        {
            r7 += r0;
        }
        r0 = r0 << 1;
    }
    return r7;
}
```

- 空间复杂度：S(1); 占用空间为常量，不随变量的值变化
- 时间复杂度：O(logn); 取决于乘数的二进制表示的位数。

3.lab02

```
short lab02(short r0)
{
    short r1 = 2, r2 = 1, r3 = 1;
    short r7 = r1;
    r0 -= 2;
    if(r0 < 0)
    {//判断是否为初始量
        r7 = 1;
        return r7;
```

```
    }
    for(; r0 != 0; r0 --)
    {
        r3 *= 2;
        r7 = r3 + r1;
        r3 = r2;
        r2 = r1;
        r1 = r7;  //更新状态
    }
    r7 = r7 % 1024;
    return r7;
}
```

- 空间复杂度：S(1); 占用空间为常量。
- 时间复杂度：O(n); 每次循环再向前一个数，n的求解需要n-3次循环，性能较差。

4.lab03

```
short lab03(short r0)
{
    short r7;
    BiTNode temp;
    BiTNode *tree;
    tree = (BiTNode*)malloc(11 * sizeof(BiTNode));
    //0为废结点
    tree[1] = {24, 706 ,0, 0};
    tree[2] = {144, 642, 1, 3};
    tree[3] = {456, 66, 0, 0};
    tree[4] = {1088, 2, 2, 6};
    tree[5] = {1092, 290, 0, 0};
    tree[6] = {2096, 898, 5, 0};
    tree[7] = {4200, 322, 4, 9};
    tree[8] = {8192, 514, 0, 0};
    tree[9] = {12000, 258, 8, 10};
    tree[10] = {14000, 898, 0, 0};
    temp = tree[7];
    while(1)
    {
        if(r7 - temp.num == 0)
        {
            r7 = temp.ans;
            return r7;
        }
        else if(r7 - temp.num < 0)
        {
            if(temp.lchild)
            {
                temp = tree[temp.lchild];
            }
            else printf("cannot find!\n");
        }
        else
        {
            if(temp.rchild)
            {
                temp = tree[temp.rchild];
            }
```

```
            else printf("cannot find!\n");
        }
    }
}
```

- 空间复杂度：S(n); 需要构建相应的二叉树，于问题规模相关。n个值要构建n个结点。
- 时间复杂度：O(logn); 使用二叉排序树，查找范围指数性缩减。

5.lab04.rec

```
short lab04_rec( int addressnum )
{
    addressnum --;
    if(addressnum == 0)
        return 1;
    else return lab04_rec(addressnum) + 1;
}
```

- 空间复杂度：S(n); 需要栈空间来支持递归，栈的最深时为递归到基本项时，性能较差。
- 时间复杂度：O(n); 每次递归，问题规模小1，递归次数正比于问题规模。

6.lab04.mod

```
short lab04_mod( short r0)
{
    short r1, r2;
    while(r0 > 7)
    {
        r1 = r0/7;
        r2 = r0 - r1*8;
        r0 = r1 + r2;
    }
    if(r0 == 7)
        return 0;
    else return r0;
}
```

- 空间复杂度：S(1); 占用空间为常量
- 时间复杂度：O(logn);每次循环将问题规模缩减一定倍数。

7.lab05

```
int lab05(short r0)
{
    int i = 2;
    short r1 = 1;
    while(i * i <= r0)
    {
        if(r0 % i == 0)
        {
            r1 = 0;
            break;
        }
        i++;
    }
```

```
    return r1;
}
```

- 空间复杂度：S(1); 占用空间为常量
- 时间复杂度：O(√n);

二、总结与思考

> 1.为什么高级语言写起来更容易?

显而易见，C语言写的代码要比汇编或者机器码写的更少，更容易让人看懂。

a) 隐藏细节

例如在rec程序中，用汇编写时需要考虑手动去保存一些寄存器的值，用C写代码时无需考虑这些。这种保存现场和恢复现场的任务代码在后续过程会自动帮你实现。而我们可以看到这种保存恢复工作的代码占据汇编代码的5行。而在某些汇编语言中占比可超过一半，如书上给的求n!的例子。

b)人性化表达

无论是汇编还是机器码，是直接与机器进行交互的。作为人的程序员读起来当然感觉不太舒服。高级语言基于与程序员互动开发，有许多人性化表达或操作。例如在使用一些变量时不在用考虑这个值放入R1还是R2，命名为number就很清楚的让人明白什么意思。

在阅读汇编时候令人头大的是各种跳转，虽然有标签可以一一对应跳转位置，但几个跳转叠在一起时，仍然让人很不舒服。而高级语言中不存在这些问题，因为它们有专门的for,while语句，结构清晰，内容明了。而且括号一括便知道循环代码在何处终止。一句

```
for(i = 0, i <= 5, i++)
{}
```

等价于

```
AGAIN AND R1 R1 #0;
      ADD R0 R0 #1;
      ADD R2 R0 #-5;
      BRp AGAIN
```

> 2.可以添加哪一条代码使得编程大大简化?

减法指令 SUB

理由：在程序设计中会涉及大量的比较，在这种情况下，基于15条指令只能对其中一个取反加一再相加来判断。如果使用减法指令可简化这一过程，使代码量减少且易读。

```
    1101 DR SR1 000 SR2
```

> 3.学习LC3对我使用高级语言有何帮助?

不丢失对细节的控制。

a) 更快找出错误

由于高级语言隐藏了太多的细节，有些错误发生的时候，很难迅速找到它，因为你不知道机器是如何执行你写下的代码，它下面的数据传输过程需要你足够了解底层原理和各部分之间的数据传输关系。就像书上的2+3 = e 这个例子，如果你不知道键盘输入的为ASCII码，不进行转换，很可能得出错误结果。

最好的例子莫过于向左旋*25*度

b) 写出效率更高的代码

一个递归程序可能是一个好的递归，也可能是一个非常坏的递归（very very bad）。书上举出的求n!，在递归的情况下时间与空间占用都极大地超过了非递归，使用递归完全好处，在这种情况下，如果知道栈的工作原理，就不会做出如此愚蠢的决定。避免写出效率低下的代码。